Part 4 – Assembly Programming Basics

Swapnil Pathak
Amit Malik

www.SecurityXploded.com

Disclaimer

The Content, Demonstration, Source Code and Programs presented here is "AS IS" without any warranty or conditions of any kind. Also the views/ideas/knowledge expressed here are solely of the trainer's only and nothing to do with the company or the organization in which the trainer is currently working.

However in no circumstances neither the trainer nor SecurityXploded is responsible for any damage or loss caused due to use or misuse of the information presented here.

Acknowledgement

- Special thanks to **null** & **Garage4Hackers** community for their extended support and cooperation.
- Thanks to all the **Trainers** who have devoted their precious time and countless hours to make it happen.

Reversing & Malware Analysis Training

This presentation is part of our **Reverse Engineering & Malware Analysis** Training program. Currently it is delivered only during our local meet for FREE of cost.

For complete details of this course, visit our Security Training page.

Who am I #1

Amit Malik (sometimes DouBle_Zer0,DZZ)

- Member SecurityXploded
- Security Researcher @ McAfee Labs
- RE, Exploit Analysis/Development, Malware Analysis
- Email: m.amit30@gmail.com

Who am I #2

Swapnil Pathak

- Member SecurityXploded
- Security Researcher @ McAfee Labs
- RE, Malware Analysis, Network Security
- Email: swapnilpathak101@gmail.com

Course Q&A

- Keep yourself up to date with latest security news
  - http://www.securityphresh.com
- For Q&A, join our mailing list.
  - http://groups.google.com/group/securityxploded

Presentation Outline

- Intro to x86-32
- Assembly Language
- Instructions
- Stack Operations
- Calling conventions
- Demo

x86-32

- 32 bit instruction set architectures based on Intel 8086 CPU
- Address a linear address space up to 4GB
- 8, 32 bit General Purpose Registers (GPR)
- 6,16 bit Segment Registers
- EFLAGS and EIP register
- Control Registers (CR0-CR4) (16 bits)
- Memory Management Registers Descriptor Table Registers (GDTR, IDTR, LDTR)
- Debug Registers (DR0-DR7)

Registers Usage - RE

- Register
  - Storage Locations.
  - Much faster access compare to memory locations.
- EAX: Accumulator , mostly stores return values from functions (APIs)
- EBX: Base index (for use with arrays)
- ECX: Counter
- EDX: Data/general
- ESI: *Source index* for string operations.

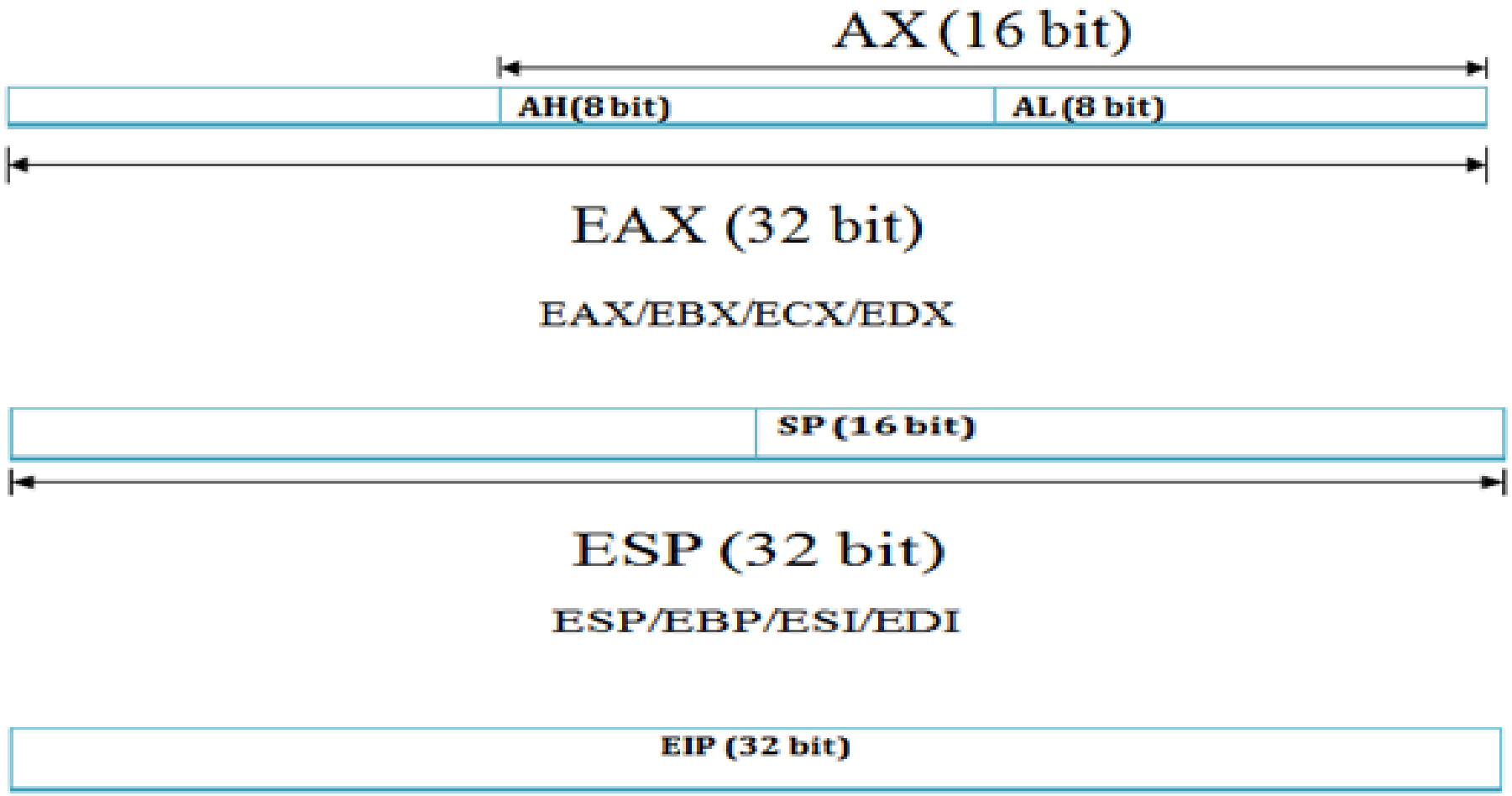
Registers Usage – RE Cont.

- EDI: *Destination index* for string operations.
- ESP: Stack pointer for top address of the stack.
- EBP: Stack base pointer for holding the address of the current stack frame.
- EIP: Instruction pointer. Holds the program counter, the next instruction address.
- Segment registers:
  - Used to address particular segments of memory (code, data, stack)

    !) CS: Code !!) SS: Stack

    !!!) ES: Extra !V) DS: Data V) FS, GS

Registers – 32 bit (X86)



(R/E)Flags Register

- Bit field of states
- Status Flags
  - Carrry (CF) : set when an arithmetic carry/borrow has been generated out of the MSB.
  - Zero (ZF) : set when an arithmetic operation result is zero and reset otherwise.
  - Sign (SF) : set when an arithmetic operation set the MSB i.e. the result value was negative.
  - Trap (TF) : when set permits operation of processor in single-step. Mostly used by debuggers.
  - Interrupt (IF) : determines whether the CPU should handle maskable hardware interrupts.
  - Direction (DF) : determines the direction (left-to-right or right-to-left) of string processing.
  - Overflow (OF) : indicates arithmetic overflow.

Assembly Language

- Low level programming language
- Symbolic representation of machine codes, constants.
- Assembly language program consist of sequence of process instructions and meta statements
- Assembler translates them to executable instructions that are loaded into memory and executed.
- Basic Structure

  [label] : opcode operand1, operand2

  opcode – mnemonic that symbolize instructions
- Example.
  - **MOV** AL, 61h => 10110000 01100001

Instructions

ADD dst, src

- Adds the values of src and dst and stores the result into dst.
- For example ADD EAX, 1

SUB dst, src

- Subtracts src value from dst and stores the result in dst.
- For example SUB EAX, 1

CMP dst, src

- Subtracts src value from dst but does store the result in dst
- Mostly used to set/reset decision making bits in EFLAGS register such as ZF
- For example CMP EAX, EBX

Instructions cont.

MOV dst, src

- Moves data from src (left operand) to destination (right operand)
- For example mov EDI, ESI

Note :

- Both operands cannot be memory locations.
- Both the operands must be of the same size

LEA dst, src

- Stands for Load Effective Address.
- Computes the effective address of src operand and stores it in dst operand.
- For example LEA ECX,[EBX + 5]

Note:

- Generally brackets denote value at memory locations.
- In case of LEA it does simple arithmetic and stores it in dst

Instructions cont.

XOR dst, src

- Performs a bitwise exclusive OR operation on the dst and src and stores the result in dst.
- Each bit of the result is 1 if the corresponding bits of the operands are different, 0 if the corresponding bit are same

Note :

- When used with same register clears the contents of the register
- Optimized way to clear the register. Better than MOV EAX, 0

Instructions cont.

REP

- Used with string operations
- Repeats a string instruction until ECX (counter register) value is equal to zero.
- For example REP MOVS byte ptr DS:[EDI], DS:[ESI]

LOOP

- Similar to loops in high level languages
- Used to execute sequence of instructions multiple times.
- For example

```
MOV ECX, 10
Test : INC EBX
       INC EAX
       LOOP  Test
```

Instructions cont.

TEST dst, src

- Performs bitwise logical and between dst and src
- Updates the Zero flag bit of the EFLAGS register
- Mostly used to check if the return value of the function is not zero
- For example TEST EAX, EAX

INT 3h

- Breakpoint instruction
- Used by debuggers to stop execution of the program at particular instruction

Instructions cont.

CALL address

- Performs two functions
  - Push address of the next instruction on stack (return address)
  - Jump to the address specified by the instruction
- For example CALL dword ptr [EAX+4]

RET

- Transfers the control to the address previously pushed on the stack by CALL instruction
- Mostly denotes the end of the function

Instructions cont.

Jump instructions

- Categorized as conditional and unconditional
- Unconditional jump instructions
  - JMP (Far Jump) – E9 – (Cross segments)
  - JMP (Short Jump) – EB – (-127 to 128 bytes)
  - JMP (Near Jump) – E9 – (in a segment)
- For example JMP EAX

- Conditional jump instructions
  - Jumps according to bit flags set in the EFLAGS register
  - JC, JNC, JZ, JNZ, JS, JNS, JO, JNO
  - Unsigned comparisons JA, JAE, JB, JBE
  - Signed comparisons JG, JGE, JL, JLE
  - Usually followed by CMP instruction

Instructions cont.

PUSH operand

- Pushes operand on the stack
- Decrements the stack pointer register by operand size
- For example PUSH EAX

POP operand

- Stores the value pointed by the stack pointer in operand
- Increments the stack pointer register by operand size
- For example POP EAX

Note: POP/PUSH EIP is an invalid instruction

PUSHF, POPF

Calling Conventions

- Describes how the arguments are passed and values returned by functions.
- Steps performed when a function is called
  - Arguments are passed to the called function
  - Program execution is transferred to the address of the called function
  - Called function starts with lines of code that prepare stack and registers for use within the function. Also known as **function prologue**.
    - For e.g.

      ```
      push ebp
      mov ebp, esp
      or with enter instruction
      ```
  - Called function ends with lines of code that restore stack and registers set initially. Also known as **function epilogue**.
    - For e.g.

      ```
      mov esp, ebp
      pop ebp
      ret
      or with leave instruction
      ```
  - Passed arguments are removed from the stack, known as stack cleanup. Can be performed by both calling function or called function depending on the **calling convention used.**

Calling conventions cont.

- __cdecl (C calling convention)
  - Arguments are passed from right to left and placed on the stack
  - Stack cleanup is performed by the caller
  - Return values are stored in EAX register
  - Standard calling convention used by C compilers

- __stdcall (Standard calling convention)
  - Arguments are passed from right to left and placed on the stack
  - Stack cleanup is performed by the called function
  - Return values are stored in EAX register
  - Standard calling convention for Microsoft Win32 API

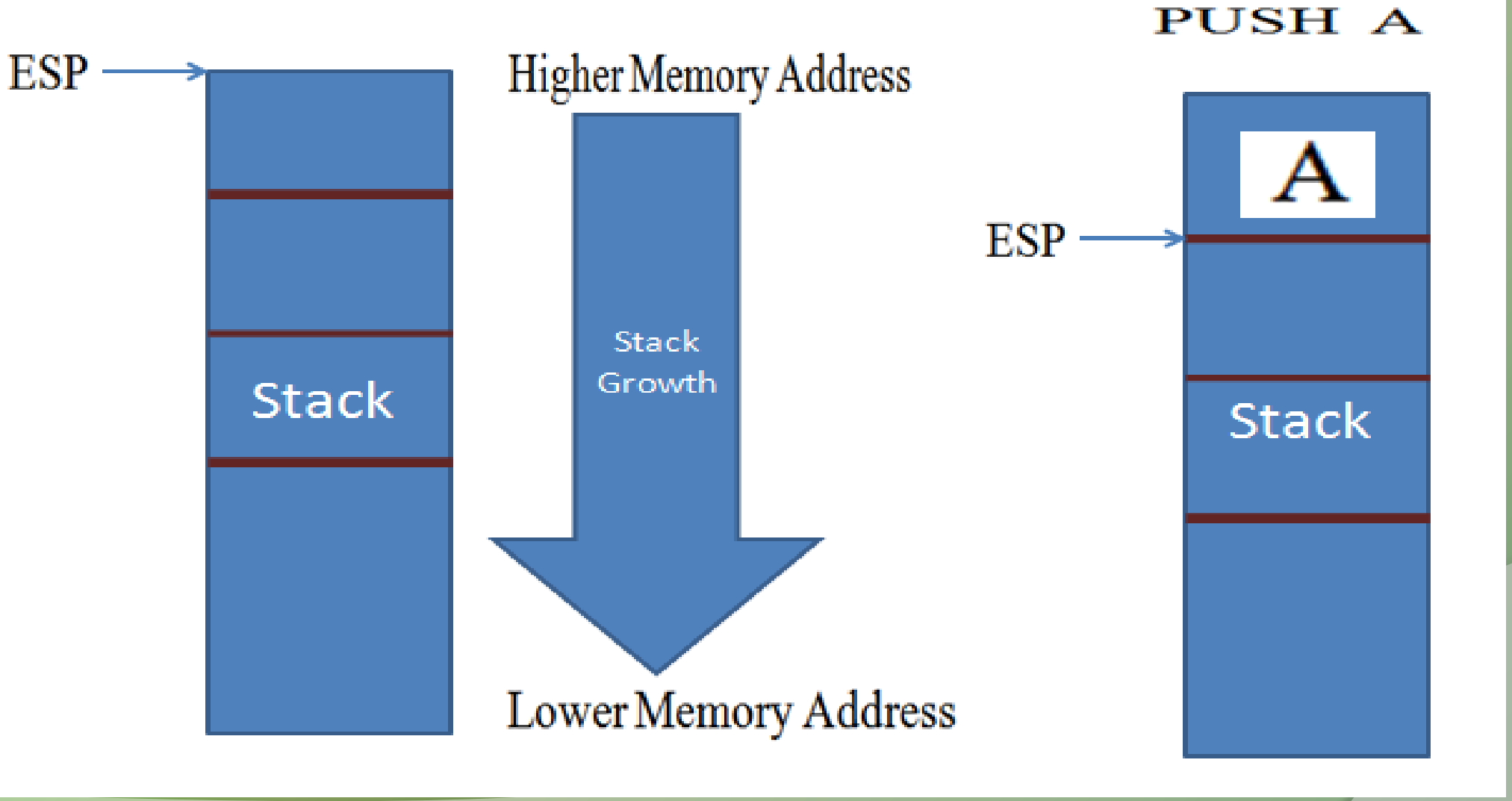
- __fastcall (Fast calling convention)
  - Arguments passed are stored in registers for faster access

- Thiscall
  - Arguments are passed from right to left and placed on the stack. this pointer placed in ECX
  - Standard calling convention for calling member functions of C++ classes

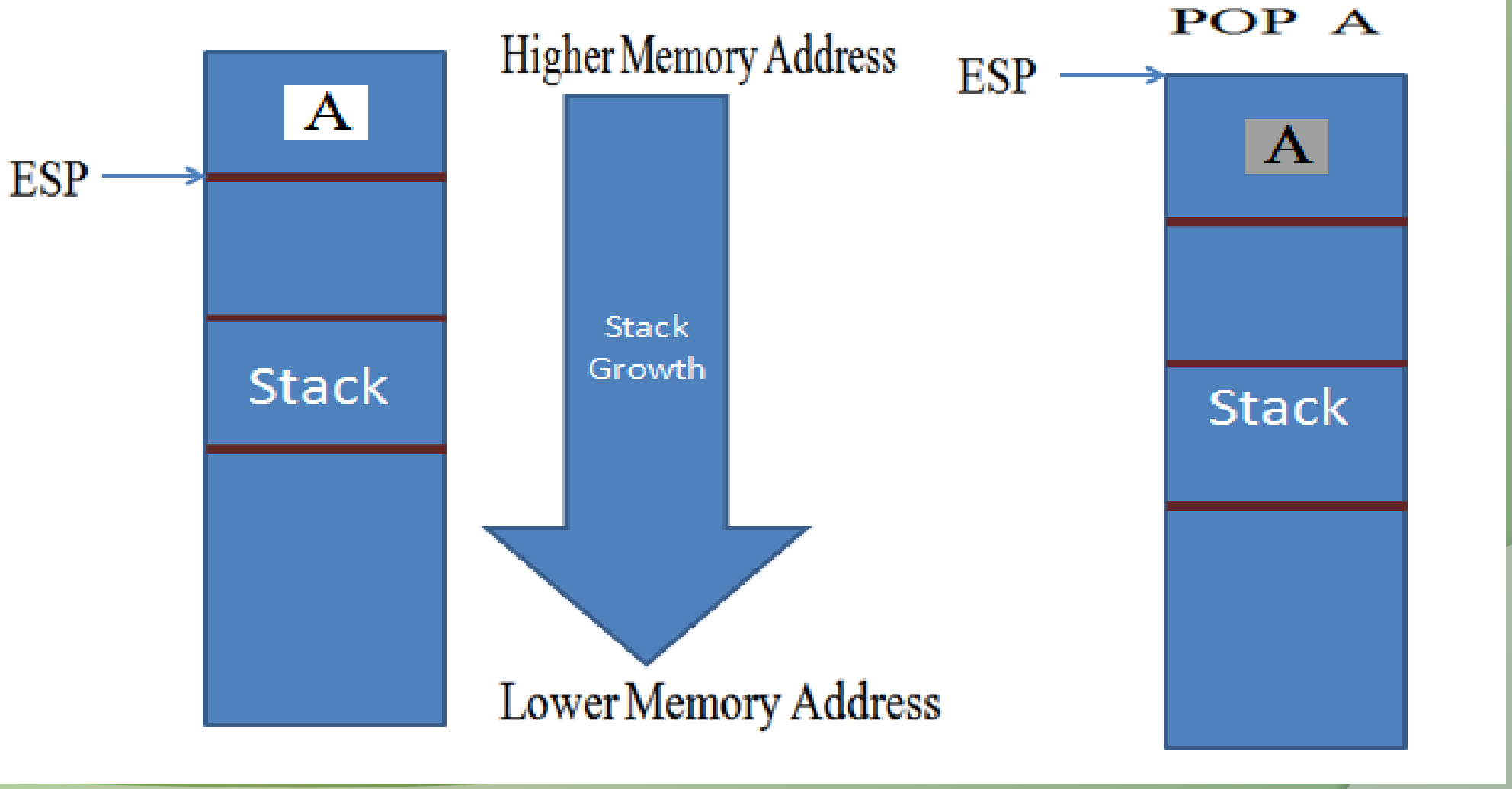
Stack operations

- Stack is a LIFO (Last In First Out) type data structure
- Stacks grows downward in memory, from **higher memory** address to **lower memory** address
- PUSH decrement the stack pointer i.e ESP
- POP Increment the stack pointer i.e ESP
- **Each function has its own stack frame**
- Function prologue setup the stack frame for each function
- **Local variable of a function are stored into its stack frame**

Stack #1


ESP
Stack
Higher Memory Address
Stack
Growth
Lower Memory Address
PUSH A
A
ESP
Stack

Stack #2


A
ESP
Stack
Higher Memory Address
Stack
Growth
Lower Memory Address
POP A
ESP
A
Stack

Stack #3

- Each function creates its own stack.
- Caller function stack: known as parent stack.
- Called function stack: known as child stack.

For e.g.

```
main(){
        sum();
}
```

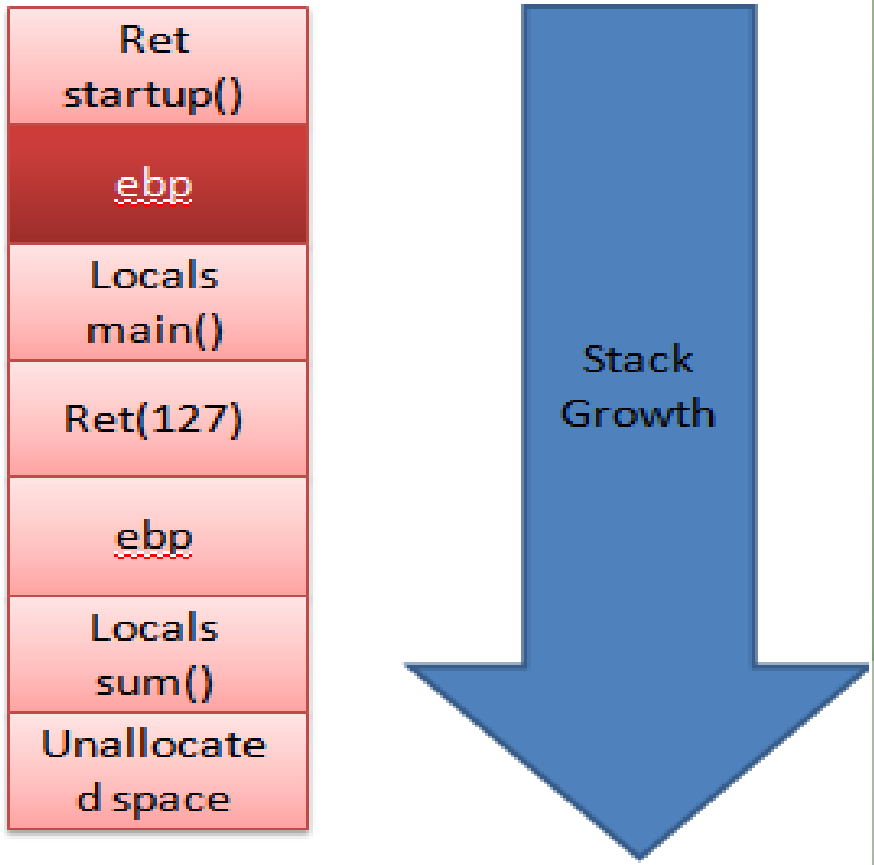
ASM Pseudo:

```
_main:
123:    push ebp
124:    mov ebp,esp
125:    sub esp,val
126:    call _sum
127:    mov esp,ebp
128:    pop ebp
129:    ret
```

* The parent and child notation is the instructor notation, technically it should be caller and callee stack frames.

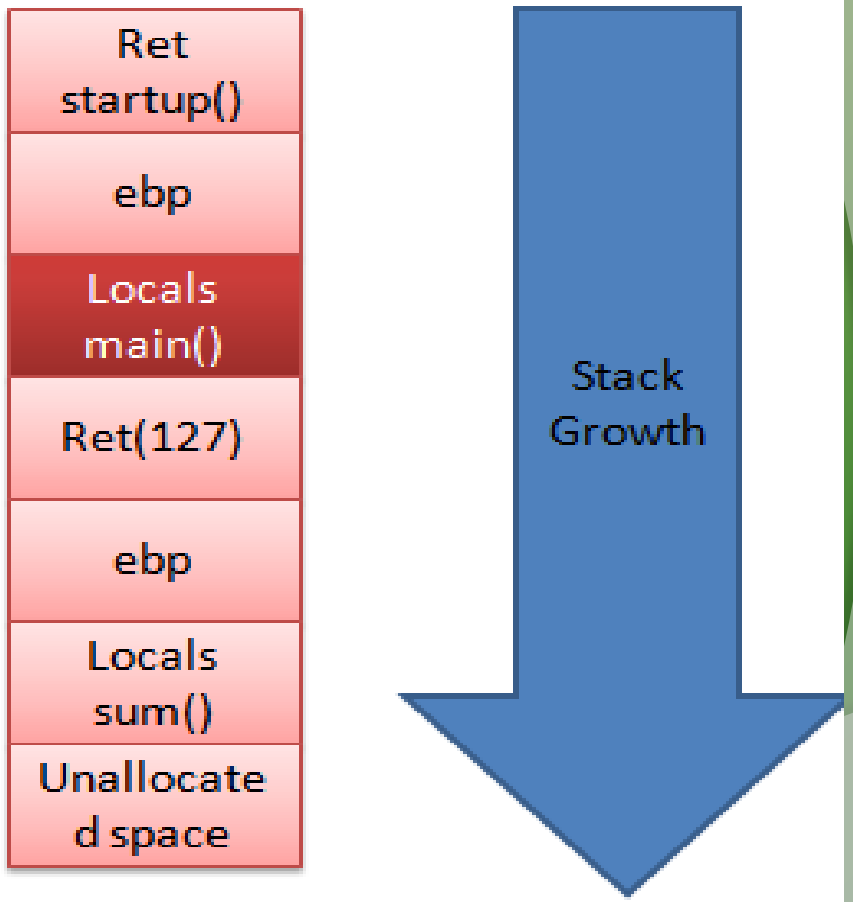
Stack #4

```
123:    push ebp
124:    mov ebp,esp
125:    sub esp,val
126:    call _sum
127:    mov esp,ebp
128:    pop ebp
129:      ret
```



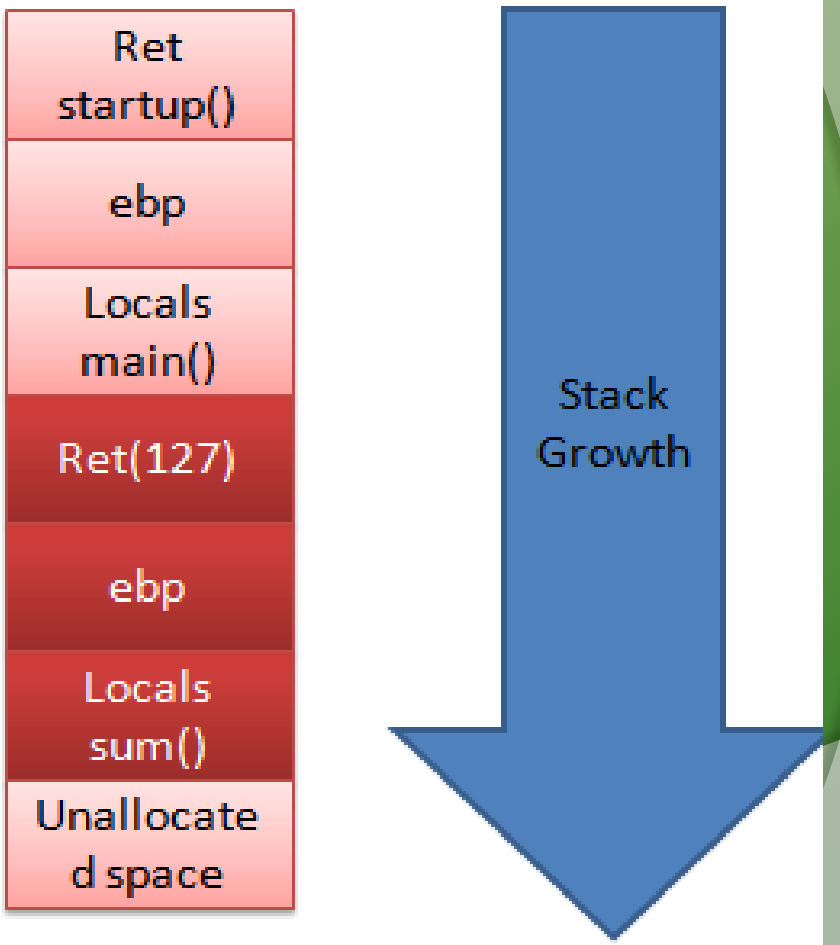
Stack #5

```
123:    push ebp
124:    mov ebp,esp
125:    sub esp,val
126:    call _sum
127:    mov esp,ebp
128:    pop ebp
129:     ret
```



Stack #6

```
123:    push ebp
124:    mov ebp,esp
125:    sub esp,val
126:    call _sum
127:    mov esp,ebp
128:    pop ebp
129:     ret
```



DEMO (Source Code)

```
#include <stdio.h>
/*
Author: Amit Malik
http://www.securityxploded.com    - Compile in Dev C++
*/
int mysum(int,int);
int main()
{
   int a,b,s;
   a = 5;
   b = 6;
   s = mysum(a,b);        // call mysum function
   printf("sum is: %d",s);
   getchar();
}
int mysum(int l, int m)   // mysum function
{
   int c;
   c = l + m;
   return c;
}
```

DEMO (Video)

http://vimeo.com/36198403

x86-64 Intro.

- 64 bit instruction set architectures based on Intel 8086 CPU
- Address a linear address space up to 16TB
- 16, 64 bit General Purpose Registers (GPR)
- 6, 16 bit Segment Registers
- RFLAGS and RIP register
- Control Registers (CR0-CR4) and CR8 (16 bits)
- Memory Management Registers Descriptor Table Registers (GDTR, IDTR, LDTR) size expanded to 10 bytes
- Debug Registers (DR0-DR7)

Reference

- Complete Reference Guide for Reversing & Malware Analysis Training

Thank You !

www.SecurityXploded.com